

# 第5章 VC目标代码的阅读理解

---

**\*5.1 汇编语言形式的目标代码**

**\*5.2 C语言部分编译的解析**

**\*5.3 C++部分功能实现细节**

5.4 目标程序的优化

\*5.5 C库函数分析

\*5.6 C程序的目标代码分析

# 汇编语言形式的目标代码

---

- 符号化表示
- 类型的转换
- 表达式求值
- 指针的本质
- 引用的实质

# \*符号化表示

## ➤演示函数cf52源代码

函数cf52的功能：  
计算两个整数之差的绝对值

2个参数

```
int  cf52(int parx, int pary)
{
    int  varz;
    varz = parx - pary;
    if ( varz < 0 )
        varz = - varz;
    return  varz;
}
```

1个局部变量

# 符号化表示

## ► 演示函数cf52目标代码

禁用编译优化

`_TEXT SEGMENT`

段 `_TEXT`

```
_varz$ = -4 ;size = 4  
_parx$ = 8 ;size = 4  
_pary$ = 12 ;size = 4
```

常量符号

`?cf52@@YAHHH@Z PROC ;cf52`

`;2 : {`

`push ebp`

`mov ebp, esp`

`push ecx`

`;3 : int varz;`

`;4 : varz = parx - pary;`

`mov eax, DWORD PTR _parx$[ebp]`

`sub eax, DWORD PTR _pary$[ebp]`

`mov DWORD PTR _varz$[ebp], eax`

对应函数cf52的过程

对应源代码

ASM YJW

# 符号化表示

## ➤ 演示函数cf52目标代码（续）

禁用编译优化

```
;5 :          if ( varz < 0 )
jns     SHORT $LN1@cf52
; 6 :          varz = - varz;
mov     ecx, DWORD PTR _varz$[ebp]
neg     ecx
mov     DWORD PTR _varz$[ebp], ecx
$LN1@cf52:
; 7 :          return varz;
mov     eax, DWORD PTR _varz$[ebp]
; 8 :      }
mov     esp, ebp
pop     ebp
ret     0
```

对应函数cf52的过程 之结束

```
?cf52@@YAHHH@Z     ENDP          ; cf52
```

```
_TEXT     ENDS
```

段 \_TEXT 之结束

# 符号化表示

## ➤ 符号常量与堆栈的位置关系

```
_varz$  = -4           ;size = 4  
_parx$  = 8            ;size = 4  
_pary$  = 12           ;size = 4
```

```
?cf5200@YAHOOZ  PROC    ; cf52  
;2 : {  
push    ebp  
mov     ebp, esp  
push    ecx  
;3 :      int    varz;  
;4 :      varz = parx - pary;  
mov     eax, DWORD PTR _parx$[ebp]  
sub     eax, DWORD PTR _pary$[ebp]  
mov     DWORD PTR _varz$[ebp], eax
```

堆栈底部



# \*类型的转换

---

## ➤关于C语言的类型转换

- ✓ 在C语言中，有自动类型转换和强制类型转换两种情形。
- ✓ 在计算算术表达式时，要求操作数的数据类型一致，如果不一致，低精度操作数被自动转换为高精度操作数。在计算整型算术表达式时，至少采用整型类型的精度，如果表达式中有字符型操作数，那么在使用之前被自动转换为整型类型。
- ✓ 可以根据需要，采用强制类型转换的方式，明确要求实施类型转换。

# 类型的转换

## ➤ 演示函数cf54

演示自动类型转换和强制类型转换

```
char cf54(int para, int parb)
{
```

```
    int dvar;
```

1个整型局部变量

```
    dvar = ( char ) ( 13 * para ) + 19 * ( char ) parb;
```

```
    return dvar;
```

```
}
```

2个整型参数

自动类型转换

强制类型转换

强制类型转换

# 类型的转换

## ➤ 演示函数cf54目标代码

禁用编译优化

```
push    ebp
mov     ebp, esp
push    ecx
        ; dvar = ( char )( 13*para ) + 19*( char )parb;
mov     eax, DWORD PTR _para$[ebp]
imul    eax, 13
movsx   ecx, al
movsx   edx, BYTE PTR _parb$[ebp]
imul    edx, 19
add     ecx, edx
mov     DWORD PTR _dvar$[ebp], ecx
mov     al, BYTE PTR _dvar$[ebp]
mov     esp, ebp
pop     ebp
ret
```

自动  
类型  
转换

强制类型转换(char)(13\*para)

强制类型转换(char) parb

强制类型转换, 返回类型char

# \*表达式求值

---

## ➤关于C语言中部分运算符

- ✓ C语言中，对四个运算符的操作数求值顺序有明确规定。
- ✓ **逻辑与运算符**：先对左侧操作数进行求值，如果值为真，再对右侧操作数进行求值。如果左侧操作数的值为假，不对右侧操作数进行求值。
- ✓ **逻辑或运算符**：先对左侧操作数进行求值，如果值为真，不再对右侧操作数进行求值。
- ✓ **条件运算符**：先对表达式1进行求值，如果值为真，仅对表达式2进行求值。如果表达式1的值为假，仅对表达式3进行求值。
- ✓ **逗号运算符**：从左到右依次对各操作数进行求值

# 表达式求值

## ➤ 演示函数cf56

### 演示表达式求值

```
int cf56(int para, int parb)
{
    int m, n, x;
    m = n = 0;
    x = ( para >= parb ) ? ( m = 1 ) : ( n = 2 );
    x += ( para <= parb ) || ( m += 10 , n += 20 );
    x += ( para != parb ) && ( m += 100 , n += 200 );
    return x + m + n;
}
```

# 表达式求值

禁用编译优化

## ➤ 演示函数cf56的部分目标代码

```
; x = ( para >= parb ) ? ( m = 1 ) : ( n = 2 );
```

```
mov    ecx, DWORD PTR _para$[ebp]
```

```
cmp    ecx, DWORD PTR _parb$[ebp]
```

```
j1     SHORT LN3@cf56
```

判断 表达式1

```
mov    DWORD PTR _m$[ebp], 1
```

```
mov    edx, DWORD PTR _m$[ebp]
```

```
mov    DWORD PTR tv65[ebp], edx
```

```
jmp    SHORT LN4@cf56
```

计算 表达式2

LN3@cf56:

```
mov    DWORD PTR _n$[ebp], 2
```

```
mov    eax, DWORD PTR _n$[ebp]
```

```
mov    DWORD PTR tv65[ebp], eax
```

计算 表达式3

LN4@cf56:

```
mov    ecx, DWORD PTR tv65[ebp]
```

```
mov    DWORD PTR _x$[ebp], ecx
```

赋值 (表达式)

M YJW

# 表达式求值

禁用编译优化

## ➤ 演示函数cf56的部分目标代码

```
; x += ( para <= parb ) || ( m += 10 , n += 20 );
```

```
mov     edx, DWORD PTR _para$[ebp]
cmp     edx, DWORD PTR _parb$[ebp]
jle     SHORT LN5@cf56
```

判断 左侧逻辑值

```
mov     eax, DWORD PTR _m$[ebp]
add     eax, 10
mov     DWORD PTR _m$[ebp], eax
mov     ecx, DWORD PTR _n$[ebp]
add     ecx, 20
mov     DWORD PTR _n$[ebp], ecx
jne     SHORT LN5@cf56
```

判断 右侧逗号表达式

```
mov     DWORD PTR tv70[ebp], 0
jmp     SHORT LN6@cf56
```

“假”送到临时变量

LN5@cf56:

# 表达式求值

禁用编译优化

## ➤ 演示函数cf56的部分目标代码

```
; x += ( para <= parb ) || ( m += 10 , n += 20 );  
; .....
```

LN5@cf56: (续上页)

```
mov     DWORD PTR tv70[ebp], 1
```

“真”送到临时变量

LN6@cf56:

```
mov     edx, DWORD PTR _x$[ebp]  
add     edx, DWORD PTR tv70[ebp]  
mov     DWORD PTR _x$[ebp], edx
```

计算并赋值

# \*指针的本质

---

## ➤关于C语言的指针

- ✓ 指针的本质就是地址。指针变量的值应该是存储单元的地址。
- ✓ 所谓指针变量p指向变量x，实际上就是变量p含有变量x所在存储单元的地址。
- ✓ 在VC2010环境中，地址是32位的段内偏移，所以指针变量本身占用4个字节的存储单元，这与整型变量一样。
- ✓ 常常把“指针变量”简称为“指针”。

# 指针的本质

## ➤ 演示函数cf58

演示指针的本质

假设在调用函数cf58时，实参指向整型数组，该数组至少含有3个元素。

```
int cf58(int *pit)
```

```
{
```

1个指针参数

```
    int s = 0;
```

```
    s += *(pit++); //累加第0个元素值，并指向下一个元素
```

```
    s += *(++pit); //累加第2个元素值
```

```
    s += (*pit)++; //累加第2个元素值，第2个元素值增加1
```

```
    s += ++(*pit); //第2个元素值增加1，并累加之
```

```
    return s;
```

```
}
```

# 指针的本质

## ► 演示函数cf58的目标代码（部分）

禁用编译优化

;s = 0;

mov     DWORD PTR \_s\$[ebp], 0

;s += \*(pit++);

mov     eax, DWORD PTR \_pit\$[ebp]     ;EAX = pit

mov     ecx, DWORD PTR \_s\$[ebp]     ;ECX = s

add     ecx, DWORD PTR [eax]     ;ECX = s + \*pit

mov     DWORD PTR \_s\$[ebp], ecx     ;s = EAX

mov     edx, DWORD PTR \_pit\$[ebp]     ;EDX = pit

add     edx, 4     ;EDX = EDX + 4

mov     DWORD PTR \_pit\$[ebp], edx     ;pit = EDX

\_s\$ = -4

\_pit\$ = 8

s += (\*pit)

pit++

# 指针的本质

---

## ➤ 演示函数 **cf58** 的目标代码（部分）

```
;s += *(++pit);
```

```
mov     eax, DWORD PTR _pit$[ebp]
```

```
add     eax, 4
```

```
mov     DWORD PTR _pit$[ebp], eax
```

**\_s\$ = -4**  
**\_pit\$ = 8**

**++pit**

---

```
mov     ecx, DWORD PTR _pit$[ebp]
```

```
mov     edx, DWORD PTR _s$[ebp]
```

```
add     edx, DWORD PTR [ecx]
```

```
mov     DWORD PTR _s$[ebp], edx
```

**s += (\*pit)**

# 指针的本质

---

## ► 演示函数**cf58**的目标代码（部分）

```
;s += (*pit)++;
```

```
mov     eax, DWORD PTR _pit$[ebp]
```

```
mov     ecx, DWORD PTR _s$[ebp]
```

```
add     ecx, DWORD PTR [eax]
```

```
mov     DWORD PTR _s$[ebp], ecx
```

**\_s\$ = -4**  
**\_pit\$ = 8**

**s += \*pit**

```
mov     edx, DWORD PTR _pit$[ebp]
```

```
mov     eax, DWORD PTR [edx]
```

```
add     eax, 1
```

```
mov     ecx, DWORD PTR _pit$[ebp]
```

```
mov     DWORD PTR [ecx], eax
```

**(\*pit)++**

# 指针的本质

## ► 演示函数cf58的目标代码（部分）

```
;s += ++(*pit);
```

```
mov     edx, DWORD PTR _pit$[ebp]
```

```
mov     eax, DWORD PTR [edx]
```

```
add     eax, 1
```

```
mov     ecx, DWORD PTR _pit$[ebp]
```

```
mov     DWORD PTR [ecx], eax
```

**\_s\$ = -4**  
**\_pit\$ = 8**

**++(\*pit)**

```
mov     edx, DWORD PTR _pit$[ebp]
```

```
mov     eax, DWORD PTR _s$[ebp]
```

```
add     eax, DWORD PTR [edx]
```

```
mov     DWORD PTR _s$[ebp], eax
```

**s += (\*pit)**

# 指针的本质

## ➤ 演示函数cf59

演示指向指针的指针

假设在调用函数cf59时，实参指向一个指针数组，且该指针数组的元素又指向一维整型数组。

```
int cf59(int **ppt, int i)
{
    int s = 0;
    s += *( *ppt+i );
    s += *( *(ppt+i) );
    return s;
}
```

还假设这两个数组的元素个数不小于另一个参数i的值。

//ppt[0][i]

//ppt[i][0]

# 指针的本质

## ➤ 演示函数 **cf59** 的目标代码（部分）

`;s += *( *ppt+i );`

禁用编译优化

<code>mov</code>	<code>eax, DWORD PTR _ppt\$[ebp]</code>	<code>;EAX = ppt</code>
<code>mov</code>	<code>ecx, DWORD PTR [eax]</code>	<code>;ECX = *ppt</code>
<code>mov</code>	<code>edx, DWORD PTR _i\$[ebp]</code>	<code>;EDX = i</code>
<code>mov</code>	<code>eax, DWORD PTR _s\$[ebp]</code>	<code>;EAX = s</code>
<code>add</code>	<code>eax, DWORD PTR [ecx+edx*4]</code>	<code>;EAX = s + *(*ppt+4*i)</code>
<code>mov</code>	<code>DWORD PTR _s\$[ebp], eax</code>	<code>;s = EAX</code>

# 指针的本质

## ➤ 演示函数 **cf59** 的目标代码（部分）

禁用编译优化

```
;s += *( *(ppt+i) );
```

mov ecx, DWORD PTR _i\$[ebp]	;ECX = i
mov edx, DWORD PTR _ppt\$[ebp]	;EDX = ppt
mov eax, DWORD PTR [edx+ecx*4]	;EAX = *(ppt+i)
mov ecx, DWORD PTR _s\$[ebp]	;ECX = s
add ecx, DWORD PTR [eax]	;ECX = s+*(*(ppt+i))
mov DWORD PTR _s\$[ebp], ecx	;s = ECX

# \*引用的实质

---

## ➤关于C++语言的引用

- ✓ 引用（Reference）是C++语言对C语言的重要扩充。
- ✓ 引用就是某一变量（目标）的一个别名，对引用的操作与对变量直接操作完全一样。所谓别名，即是给一个已经被命名的实体赋予另一个命名的含义。
- ✓ 引用**似乎**是一种实体的命名方法，而非一个实体。**真的吗？**

# \*引用的实质

## ➤演示程序dp515

演示引用的本质

```
int main()
{
    int test_var = 1;
    int &ref_var = test_var;
    int *poi_var = &test_var;
    int *poi_ref = &ref_var;
    //
    cout<< "poi_var = " << poi_var << endl;
    cout<< "poi_ref = " << poi_ref << endl;
    //
    test_var = 1;
    cout<< "test_var = " << test_var << endl;
    ref_var = 2;
    cout<< "test_var = " << test_var << endl;
    *poi_var = 3;
    cout<< "test_var = " << test_var << endl;
    return 0;
}
```

# \*引用的实质

## ➤ 演示程序dp515的部分目标代码

```
;int  &ref_var = test_var;  
lea   eax, DWORD PTR _test_var$[ebp]  
mov   DWORD PTR _ref_var$[ebp], eax
```

形式上是值，实际上是地址

```
;int  *poi_var = &test_var;  
lea   ecx, DWORD PTR _test_var$[ebp]  
mov   DWORD PTR _poi_var$[ebp], ecx
```

形式上和实际上，都是地址

```
;int  *poi_ref = &ref_var;  
mov   edx, DWORD PTR _ref_var$[ebp]  
mov   DWORD PTR _poi_ref$[ebp], edx
```

形式上是地址，实际上是值

# \*引用的实质

## ➤ 演示程序dp515的部分目标代码

```
;ref_var = 2;
```

```
mov    eax, DWORD PTR _ref_var$[ebp]
```

```
mov    DWORD PTR [eax], 2
```

```
;
```

形式上是直接，实际上是间接

```
;cout<< "test_var = " << test_var << endl;
```

```
;省略对应目标代码
```

```
;
```

```
;*poi_var = 3;
```

```
mov    ecx, DWORD PTR _poi_var$[ebp]
```

```
mov    DWORD PTR [ecx], 3
```

形式上和实际上，都是间接